A Knowledge-Based Method for Inferring Semantic Concepts from Visual Models of System Behavior

KEVIN L. MILLS
National Institute of Standards and Technology
and
HASSAN GOMAA
George Mason University

Software designers use visual models, such as data flow/control flow diagrams or object collaboration diagrams, to express system behavior in a form that can be understood easily by users and by programmers, and from which designers can generate a software architecture. The research described in this paper is motivated by a desire to provide an automated designer's assistant that can generate software architectures for concurrent systems directly from behavioral models expressed visually as flow diagrams. To achieve this goal, an automated designer's assistant must be capable of interpreting flow diagrams in semantic, rather than syntactic, terms. While semantic concepts can be attached manually to diagrams using labels, such as stereotypes in the Unified Modeling Language (UML), this paper considers the possibility of providing automated assistance to infer appropriate tags for symbols on a flow diagram. The approach relies upon constructing an underlying metamodel that defines semantic concepts based upon (1) syntactic relationships among visual symbols and (2) inheritance relationships among semantic concepts. Given such a metamodel, a rule-based inference engine can, in many situations, infer the presence of semantic concepts on a flow diagram, and can tag symbols accordingly. Further, an object-oriented query system can compare semantic tags on diagram instances for conformance with their definition in the metamodel. To illustrate the approach, the paper describes a metamodel for data flow/control flow diagrams used in the context of a specific software modeling method, Concurrent Object-Based Real-time Analysis (COBRA). The metamodel is implemented using an expert-system shell, CLIPS V6.0, which integrates an object-oriented language with a rule-based inference engine. The paper applies the implemented metamodel to design software for an automobile cruise-control system and provides an evaluation of the approach based upon results from four case studies. For the case studies, the implemented metamodel recognized, automatically and correctly, the existence of 86% of all COBRA semantic concepts within the flow diagrams. Varying degrees of human assistance were used to correctly identify the remaining semantic concepts within the diagrams: in two percent of the cases the implemented metamodel reached tentative classifications that a designer was asked to confirm or

Authors' addresses: K. L. Mills, NIST, 100 Bureau Drive, Building 820, Mail Stop 8920, Gaithersburg, MD 20899; email: kmills@nist.gov; H. Gomaa, Department of Information and Software Engineering, Mail Stop 4A4, 4400 University Drive, George Mason University, Fairfax, VA 22030-4444; email: hgomaa@gmu.edu

override; in four percent of the cases a designer was asked to provide additional information before a concept was classified; in the remaining eight percent of the cases the designer was asked to identify the concept.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*CASE*

General Terms: Design, Experimentation, Measurement

Additional Key Words and Phrases: Software design methods, concept classification systems, knowledge-based software engineering, semantic data modeling, visual modeling, concurrent systems

1. INTRODUCTION

Software designers use visual models, such as data flow/control flow diagrams or object collaboration diagrams, to express system behavior in a form that can be understood easily by users and by programmers, and from which designers can generate a software architecture. To generate a software architecture from such visual models, software designers typically use an analysis and design method that includes a vocabulary of semantic concepts that can be expressed through the syntax of visual models. The research described in this paper is motivated by a desire to provide an automated designer's assistant that can generate software architectures for concurrent systems directly from behavioral models expressed visually as flow diagrams [Mills 1996; Mills and Gomaa 1996]. To achieve this goal, an automated designer's assistant must be capable of interpreting flow diagrams in semantic, rather than syntactic, terms. While semantic concepts can be attached manually to diagrams using labels, such as stereotypes in the Unified Modeling Language (UML), this paper considers the possibility of providing automated assistance to infer appropriate tags for symbols on a flow diagram.

The approach described in this paper is intended to apply to analysis and design methods where the analysis method explicitly provides semantic concepts to interpret the analysis model, and where the semantic concepts can be represented using a visual modeling notation. The approach, described here for a specific analysis method and visual notation in the concurrent, real-time domain, relies upon constructing an underlying metamodel that defines semantic concepts based upon (1) syntactical relationships among visual symbols and (2) inheritance relationships among semantic concepts. Given such a metamodel, a rule-based inference engine can, in many situations, infer the presence of semantic concepts on a flow diagram, and can tag symbols accordingly. Further, an object-oriented query system can compare semantic tags on diagram instances for conformance with their definition in the metamodel.

The paper is organized as follows. Section 2 describes the motivation for this research. Section 3 provides an overview of the conceptual architecture of CODA, a concurrent designer's assistant. Section 4 discusses the approach that CODA uses for semantic interpretation of visual models. Section 5 describes in detail the design and implementation of the CODA model analyzer, which classifies syntactic elements on data flow/control flow diagrams as specific semantic concepts associated with an analysis and design method. Section 6 describes the application of CODA to the design of a concurrent system. Section 7 evaluates the results obtained when CODA was applied to classify analysis models for four different concurrent designs. Section 8 discusses the approach and its application, before drawing some conclusions in Section 9.

2. MOTIVATION

Traditionally, visual notations for modeling software structure and behavior have proven quite popular, whether for structured analysis and design [DeMarco 1978; Yourdon and Constantine 1979], Jackson System Development [Jackson 1983], or real-time structured analysis and design [Mellor and Ward 1986; Ward and Mellor 1985; Hatley and Pirbhai 1988].

The popularity of visual notations has continued to increase as software analysis and design methods have adopted object-oriented concepts [Booch 1986; 1991; Coad and Yourdon 1991; Rumbaugh et al. 1991]. In fact, the adoption of the Unified Modeling Language (UML) as a standard for modeling the structure and behavior of object-oriented software seems to have firmly established the popularity of visual notations for software design, as UML encompasses a wide-range of visual models, including class diagrams, collaboration diagrams, sequence diagrams, statecharts [Harel 1988; Harel and Gary 1996], activity diagrams, component diagrams, deployment diagrams, and package diagrams [Fowler 1997; Booch et al. 1999; Jacobson et al. 1999; Rumbaugh et al. 1999].

Beyond the design of software, visual languages are being used increasingly to select, compose, and animate computer programs without the need to resort to textual descriptions or formal languages [Santucci 1996; Haarslev and Wessel 1996; Puigsegur et al. 1996]. The widespread popularity of visual languages might in part be due to the fact that humans are naturally visual creatures who can quickly grasp the significance of patterns of symbols drawn on a page. This popularity might also be due to the fact that diagrams facilitate ready communication among users, analysts, and designers by providing a simple language around which a range of discussions can be held. While debunking these and many other unproven theories regarding visual models, Blackwell notes that “. . . the most influential reason for the growth of interest in [visual programming] languages has been the popularity of direct manipulation iconic interfaces on personal computers” [Blackwell 1996]. Although Blackwell debunks many theories about the effectiveness of visual languages, we often notice that programmers who attempt to understand an unknown program, documented only in source code, resort to constructing diagrams to model the structure of the design. Whatever their motivating appeal, lacking a certain semantic rigor, diagrams admit to a range of interpretations based on various perspectives.

Lately, attempts are being made to combine the simplicity and visual appeal of diagrams with the rigor of more formal semantics in order to support automated transformation from one form of visual software model to another [Boloix et al. 1992; Karimi and Konsynski 1988; Tsai and Ridge 1988; Lor and Berry 1991]. For example, Boloix, Sorenson, and Tremblay have augmented data flow diagrams with an underlying entity-aggregate-relationship-attribute (EARA) model, based on set theory, which defines transformation rules that can convert data flow diagrams into structure charts. Approaches such as this one result in a deterministic transformation between diagrams, based on the rules defined. In effect, the rules define a formal semantics for possible diagrams.

Approaches that define an underlying formal semantics for visual models have two main shortcomings when applied to software design. First, the initial designs scratched out in the form of visual models generally include various degrees of ambiguity. As the design process proceeds, the design becomes less and less ambiguous until it reaches code that can be successfully compiled. Formal semantic models do not generally permit ambiguity to exist within a design, at least not in a form that can be processed with a computer program. The approach proposed in this paper, and implemented within CODA, expects such ambiguity, and the underlying models are prepared to consult with the human designer as necessary to gain additional information that might help resolve ambiguities. For inexperienced designers, CODA includes default rules for resolving ambiguities without consultation. For experienced designers, CODA can request additional information needed to clarify ambiguities. When an experienced designer chooses not to provide the requested information, CODA can still make default decisions as needed to resolve ambiguity; however, the designer will be asked to confirm or override default decisions. When the designer chooses neither to confirm nor override, CODA applies the default decisions.

Second, software design often proceeds through a range of levels of abstraction as designers consider various aspects of a problem or solution. Most formal semantic definitions do not share the designer's view of multiple levels of abstraction. The approach proposed in this paper permits a designer to enter semantic concepts into the initial visual model at any of several levels of abstraction, even mixing levels of abstraction on the same diagram, and yet still permits the visual model to be analyzed by a computer program.

3. OVERVIEW OF A CONCURRENT DESIGNER'S ASSISTANT

Figure 1 depicts one view of the architecture for a concurrent designer's assistant, CODA, which was the main goal of our research [Mills 1996; Mills and Gomaa 1996]. The intent of CODA is to provide a designer with an automated assistant that can help to transform an analysis model,

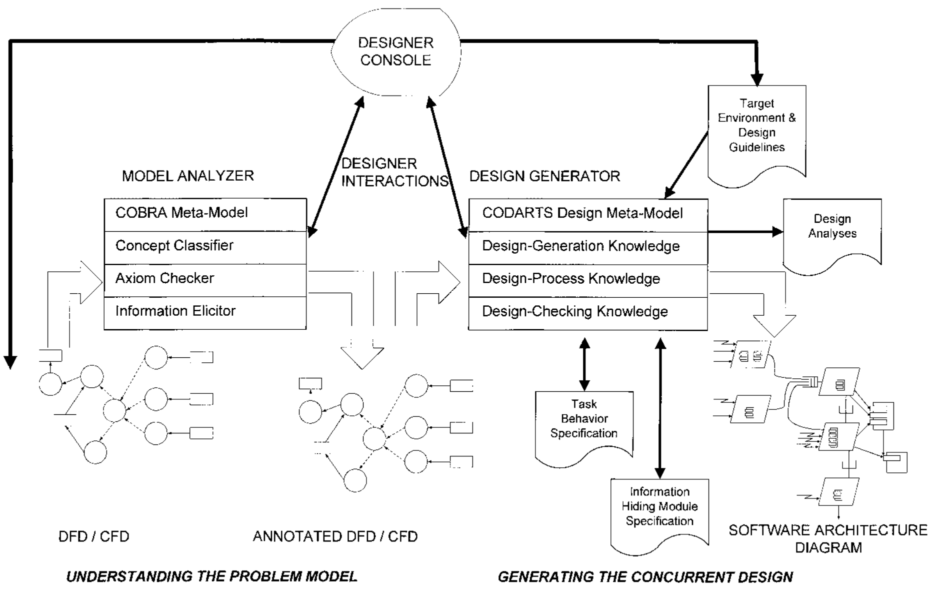

DESIGNER CONSOLE
DESIGNER INTERACTIONS
MODEL ANALYZER
COBRA Meta-Model
Concept Classifier
Axiom Checker
Information Elicitor
DESIGN GENERATOR
CODARTS Design Meta-Model
Design-Generation Knowledge
Design-Process Knowledge
Design-Checking Knowledge
Target Environment & Design Guidelines
Design Analyses
Task Behavior Specification
Information Hiding Module Specification
SOFTWARE ARCHITECTURE DIAGRAM
DFD / CFD
ANNOTATED DFD / CFD
UNDERSTANDING THE PROBLEM MODEL
GENERATING THE CONCURRENT DESIGN

Fig. 1. Conceptual Architecture for Concurrent Designer's Assistant, CODA.

typically expressed in some form of flow diagram, into a concurrent software design, expressed as a set of tasks and modules and the relationships between them. While performing this transformation, CODA also generates traceability between the analysis model and the software design, and captures the rationale for the design decisions used to accomplish the transformation. In addition, CODA can check an analysis model and a software design for conformance to the constraints imposed by a specific analysis and design method.

As shown in Figure 1, CODA consists of two main components: a model analyzer and a design generator. The model analyzer consists of four knowledge bases: (1) an analysis metamodel that describes relationships among semantic concepts within a specific analysis method, (2) concept classification rules that perform inferences on instances of semantic concepts within the analysis metamodel, (3) axioms that define relationships required and prohibited among semantic concepts in the analysis metamodel, and (4) information elicitation rules that can be used to obtain information not readily available from visual representations of the analysis metamodel. For CODA to support a specific analysis method, these four knowledge bases must be created. In the work discussed in this paper, knowledge bases were built to support Concurrent Object-Based Real-time Analysis, or COBRA [Gomaa 1993].

The design generator also consists of four knowledge bases: (1) a design metamodel that describes relationships among semantic concepts within a specific software design method, (2) heuristics codified into production rules that can transform semantic concepts from an analysis metamodel into semantic concepts within a design metamodel and that can also reason

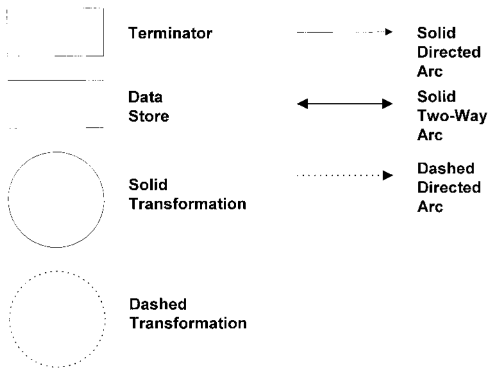


Fig. 2. COBRA syntactic elements.

about semantic concepts within a design metamodel, (3) process constraints to ensure that design decisions progress in the order required by a design method, and (4) axioms that define relationships required and prohibited among semantic concepts within the design metamodel. For CODA to support a specific design method, these four knowledge bases must be created. In the work discussed in this paper, knowledge bases were built to support Concurrent Design Approach for Real-Time Systems, or CODARTS [Gomaa 1993]. While both the model analyzer and the design generator present interesting challenges, this paper focuses mainly on the model analyzer, as specified and implemented for COBRA.

4. SEMANTIC INTERPRETATION OF VISUAL MODELS

This section describes the relationship between the syntactic elements and semantic concepts provided by COBRA, and discusses how these relationships can be represented in a metamodel for COBRA. COBRA helps a designer to model system behavior as a flow diagram, using the seven simple symbols shown in Figure 2. When used on a flow diagram, instances of these symbols can be given a name that relates to a concept in the problem domain being modeled, and some instances can be given a number that indicates a hierarchical relationship with other symbols on the diagram. To help develop a flow diagram, the COBRA analysis method provides a designer with semantic concepts, shown on Figure 3, which can be used, when reviewing natural-language requirements statements, to identify the need for elements in a flow diagram. Figure 3 implies that symbols on Figure 2 can be overloaded with several meanings. For example, a terminator can denote a device, a user role, or an external subsystem. A transformation denotes one among a range of functions or objects. A data flow can denote a system input or output or a file read or write, while a two-way data flow denotes an update. An event flow might represent one among a range of asynchronous stimuli, such as a timer, an interrupt, or a signal.

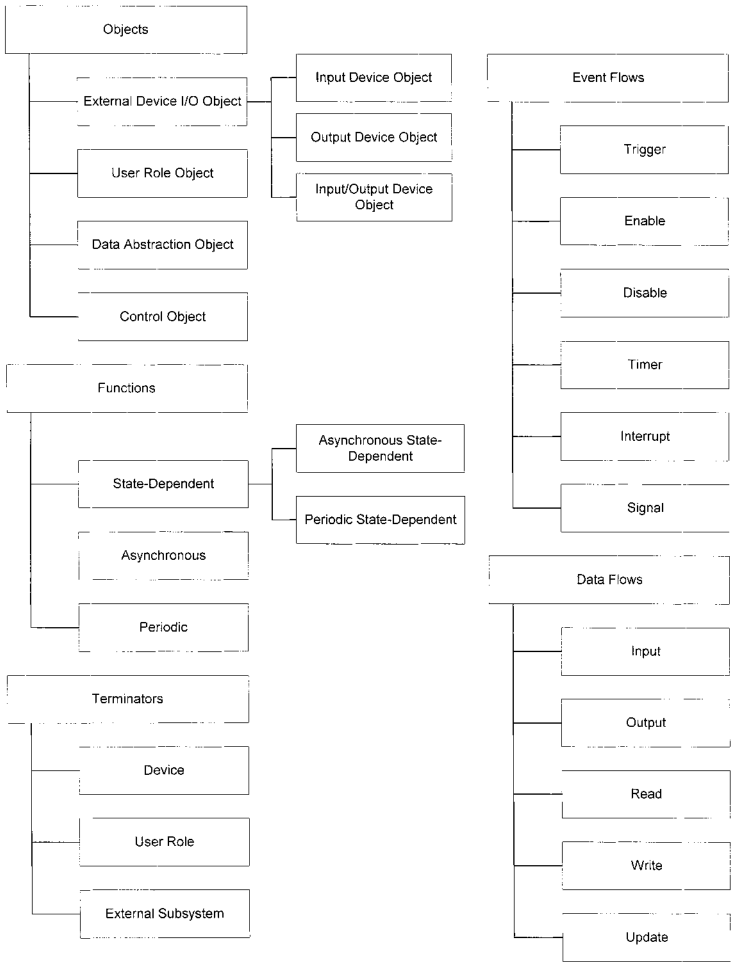


Fig. 3. COBRA semantic concepts.

4.1 Annotating COBRA Symbols with Semantic Tags

In order for the CODA design generator to perform its work, each symbol on an input flow diagram must be annotated with a semantic tag (similar to a stereotype in UML) that denotes the meaning of the symbol. For example, a circle on a flow diagram might be tagged as a device interface object or as some type of algorithm or function. Based on correct tagging, the design generator can use heuristics to consider some alternate means of mapping the circle on the flow diagram to one or more elements in a software architecture. Before passing an annotated diagram to the CODA design generator, some means must be used to verify the correctness of the semantic tags. This was originally the main purpose of the CODA model analyzer.

Recall from Figure 1 that the CODA model analyzer contains four components, two of which can be used together to verify the correctness of semantic tags on an annotated flow diagram. One component, a metamodel for COBRA, provides the semantic tags that can be assigned to symbols on the flow diagram, as well as the required relationships among those tags. (We used the COBRA modeling method instead of UML because UML did not exist when our work began.) A second component, an axiom checker, examines instances of annotated COBRA models for compliance with the COBRA metamodel. Further, we found that certain semantic concepts might be augmented with additional information that the CODA design generator can use. For example, interrupt-driven input objects might contain a maximum and expected rate at which interrupts can arrive. Given this additional knowledge, a third component, an information elicitor, can identify semantic concepts that entail added information, and, when the information is not present, can elicit the necessary data from a designer. During design of the COBRA metamodel, it became apparent that the axioms used to define each semantic concept might be converted into rules through which semantic tags on a diagram could be inferred semiautomatically.

4.2 Inferring Semantic Concepts for Tagging COBRA Models

A fourth component of the CODA model analyzer, the concept classifier, examines instances of a COBRA flow diagram and attempts to infer the semantic tags to assign to each symbol. Given that a flow diagram must be annotated with semantic tags before submission to the CODA design generator, the concept classifier can relieve a human designer from the tedious, error-prone task of labeling symbols on the diagram. Further, a human designer might label a symbol with a high-level semantic concept, such as ⟨⟨device interface object⟩⟩, which the concept classifier can convert to a more specific concept, such as ⟨⟨periodic device input object⟩⟩. When the concept classifier cannot make an unambiguous decision, the human designer can be consulted for additional information or to verify a preliminary inference or to provide a specific semantic tag. The concept classification knowledge base includes default classification rules that can be used when a designer cannot provide additional guidance. The addition of the concept classifier to the CODA model analyzer led to a series of case studies that attempted to automatically add correct semantic tags to flow diagrams.

5. DESIGN AND IMPLEMENTATION OF THE CODA MODEL ANALYZER

This section describes the technical details underlying the CODA model analyzer. The model analyzer is implemented with the CLIPS expert-system shell, and so the design and implementation are closely aligned because CLIPS provides both object-oriented and rule-based knowledge representation languages, along with an object-oriented run-time and query system, and an inference engine that can match against patterns of objects. By taking maximum advantage of these underlying execution mechanisms within CLIPS, the implementation of the CODA model analyzer is accomplished mainly by mapping concepts from a COBRA meta-

Table I. Knowledge Representation for Elements of the COBRA Metamodel and the CODA Model Analyzer

	Design Element	CLIPS Implementation Element
COBRA Meta-Model	Semantic Concept	Object Class
	Concept Taxonomy	Class-Inheritance Hierarchy
	Concept Axiom	Class-Query Specification
	Concept Classification Rule	Class-Based Expert-System Rule
CODA Model Analyzer	Concept Classification	CLIPS Inference Engine and Modules
	Classification Verification	Class Method and Subclass Checking
	Axiom Verification	Class Method and Class-Query Evaluation
	Information Elicitation	Procedure within a Function

model (described in the next subsection) directly onto CLIPS knowledge representations. The remainder of the implementation chore requires animating the CODA model analyzer by selecting appropriate CLIPS execution mechanisms for the tasks at hand: classifying or verifying concepts, checking axioms, or eliciting information from the designer. Table I shows the mapping from design element to CLIPS implementation element. These design and implementation elements are discussed below.

5.1 COBRA Metamodel

In order to apply semantic tags to the syntactic elements of a flow diagram, the CODA model analyzer encapsulates a metamodel representing the semantic concepts and relationships implied by the COBRA analysis method. A key aspect of the COBRA metamodel includes a concept taxonomy [Fikes and Kehler 1985; Lim and Cherkassky 1992], where each child concept specializes, using an **is-a** relationship, its parent concept. Figure 4 depicts part of the concept taxonomy. Each concept in the taxonomy is represented with a rectangle divided into three sections: concept name, concept attributes (if any), and concept operations. The taxonomy consists of four main layers, grouped together in Figure 4 with bounding boxes. A complete specification of the taxonomy appears elsewhere [Mills 1996].

The "Root Elements" layer provides a top for the taxonomy, along with a few traits that other concepts inherit. The next layer, "Directed Graph Elements," represents the concepts contained within directed graphs, on which flow diagrams are based. The third layer, "Syntactic Elements," denotes the symbols that can appear on a flow diagram, and defines the syntactic compositions allowed among those symbols. The bottom layer, "Semantic Concepts," organizes the semantic tags available from the COBRA analysis method. Note that concepts with italicized names, such as *Node*, represent abstract concepts that cannot be instantiated. For this reason, a flow diagram represented as a COBRA metamodel must be implemented using concepts taken from among the seven, lowest-level syntactic elements or from the semantic concepts. For brevity, many concepts are omitted from Figure 4, as denoted on the diagram with ellipses.

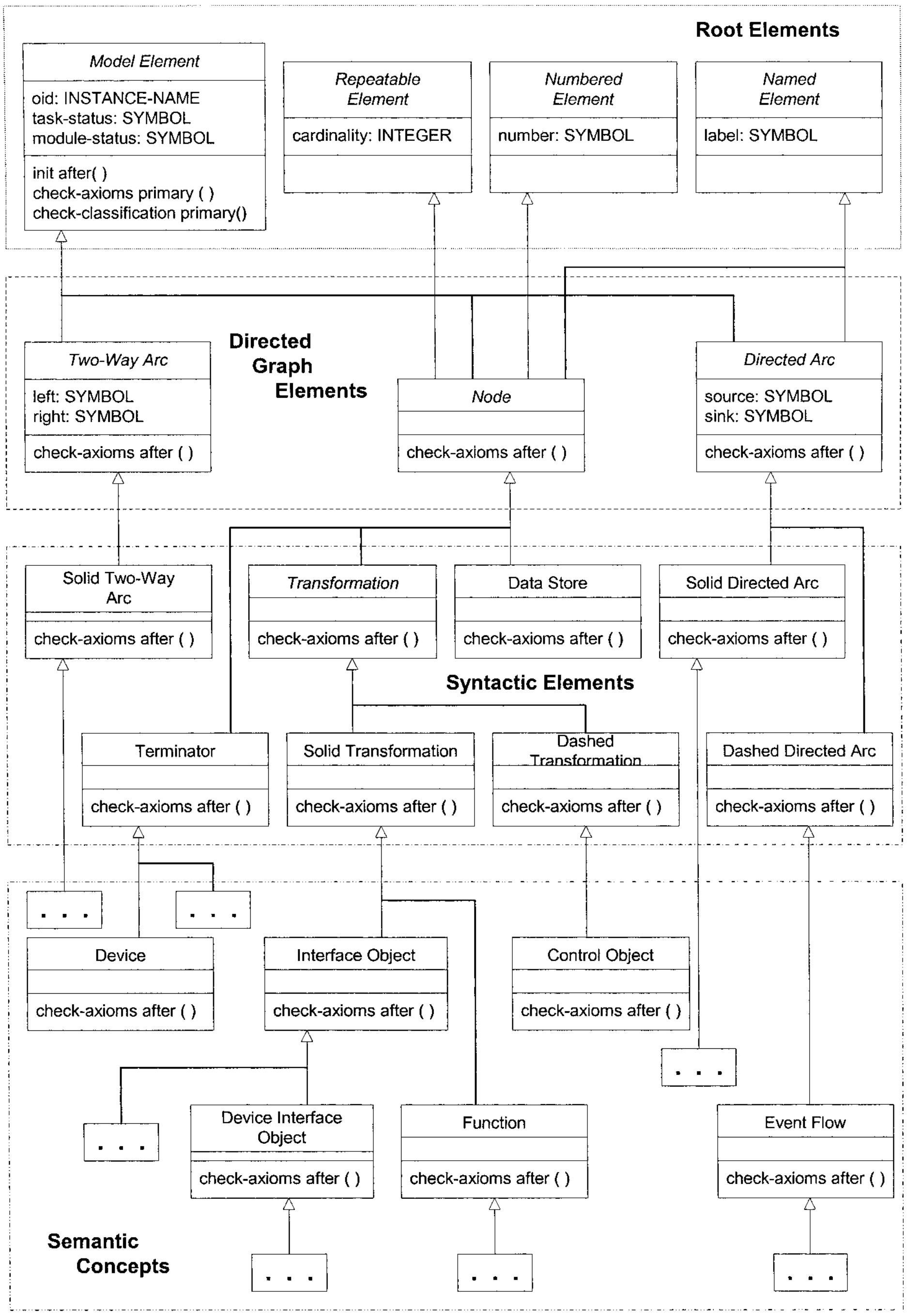


Fig. 4. The general organization of the taxonomy for the COBRA metamodel.

```
(defclass COBRA-Meta-Model::Directed-Arc
            (is-a Model-Element Named-Element)
            (role abstract)
      (slot source
            (type SYMBOL)
            (access initialize-only)
            (pattern-match reactive)
            (visibility public))
      (slot sink
            (type SYMBOL)
            (access initialize-only)
            (pattern-match reactive)
            (visibility public)))
(defmessage-handler COBRA-Meta-Model::Directed-Arc
      check-axioms after ()
        ; AXIOM BODIES OMITTED)
```

Fig. 5. CLIPS class definition for directed arc.

Each concept in the taxonomy, along with inheritance relationships, is represented directly as a CLIPS class, corresponding to a diagrammatic symbol. For example, consider the CLIPS representation for a *Directed Arc*, as shown in Figure 5. The taxonomy concept becomes a class, *Directed Arc*, within a CLIPS module, COBRA-Meta-Model. The class inherits two other classes, *Model Element* (because all symbols are model elements) and *Named Element* (because directed arcs on flow diagrams can be named). The class contains two attributes, source and sink, to hold the names of the nodes that the directed arc connects and to indicate the direction of the arc. The class also contains a single operation, **check-axioms**, which encapsulates the axioms that define instances of the class. Axioms are discussed in Section 5.4.

Given a CLIPS representation of the COBRA metamodel, a flow diagram is translated to a CLIPS representation that can be interpreted by the CODA model analyzer. For example, consider the small flow diagram fragment shown in Figure 6. Each of the 10 symbols on the diagram is represented as an instantiated CLIPS class that corresponds to that symbol. The only added information is a unique object identifier, such as **[is60]**, which is required by CLIPS to distinguish between instances of the same class. For the case studies discussed in this paper, we translated the flow diagrams manually into CLIPS representation using the technique shown in Figure 6.

While the taxonomy provides the framework for the COBRA metamodel, two other forms of knowledge, axioms and classification rules [Hayes-Roth 1985; Michalski 1980], are also required. Figure 7 shows the relationship between the classes in the taxonomy and the supporting axioms and classification rules. Private operations in each class identify specific axioms called by the check-axioms method of the class. All axioms assigned to a class must be satisfied if an object is a proper instance of the class. More particularly, a well-formed instance of a class must also satisfy all axioms for all superclasses in all its inheritance paths. Attached to selected

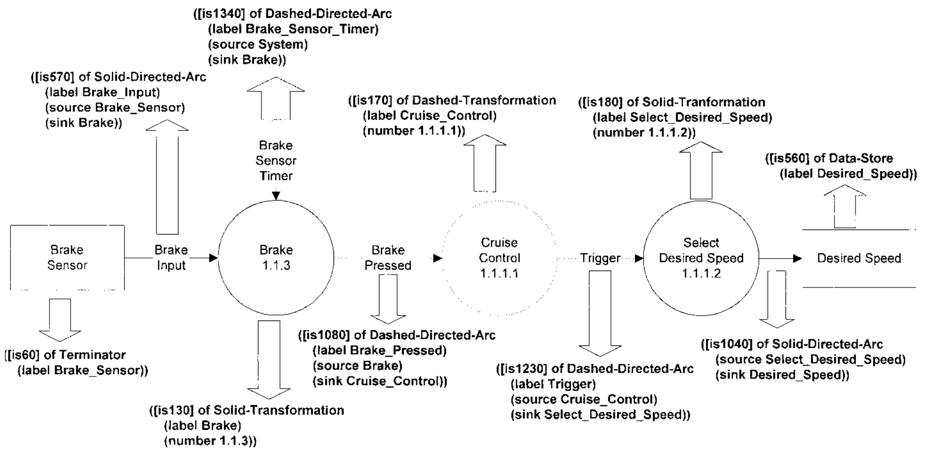


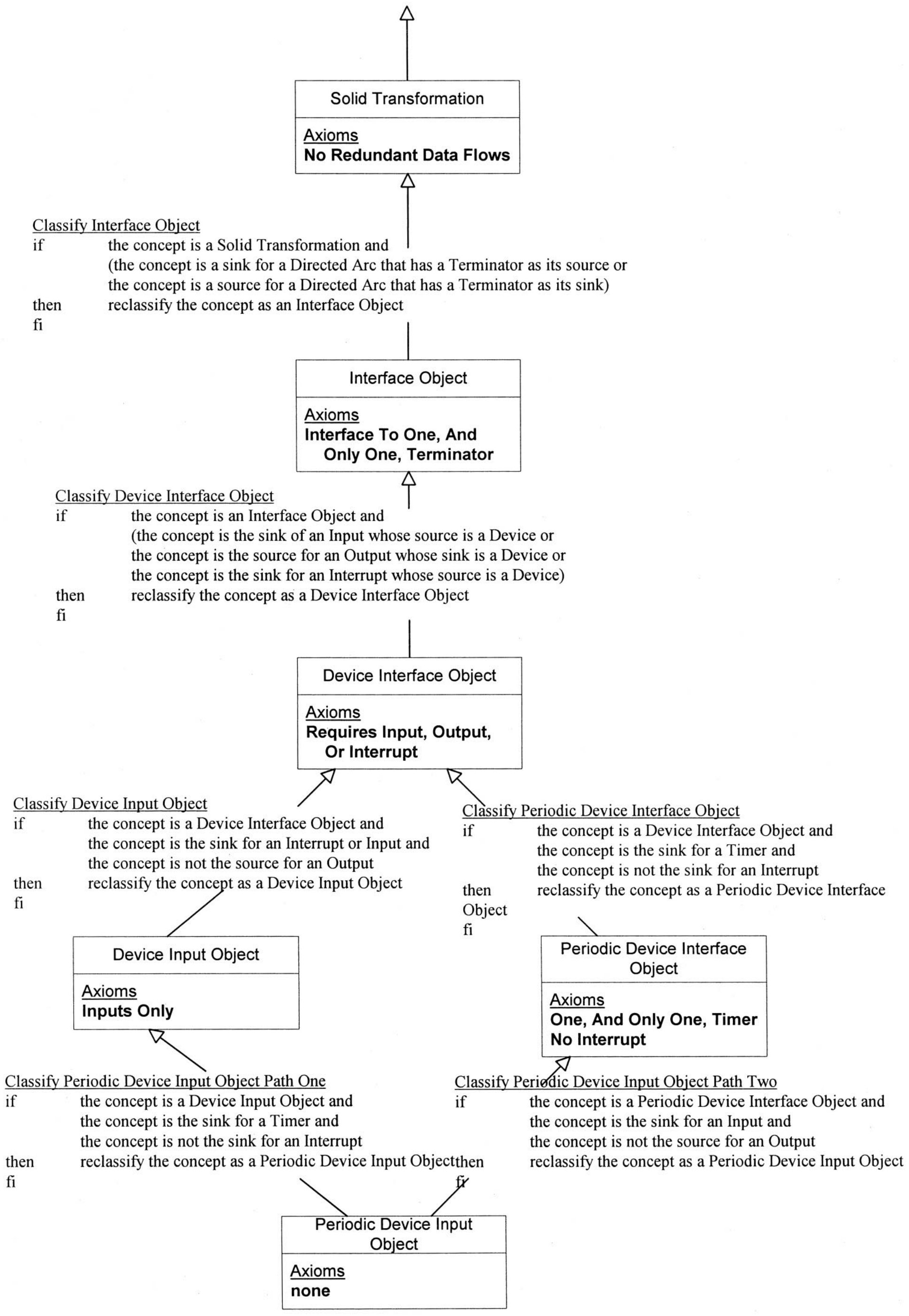
Fig. 6. Example CLIPS representation for a flow diagram.

inheritance paths in Figure 7 are classification rules. If a classification rule is satisfied on a specific inheritance path, then an instance of a superclass can become an instance of the subclass along that path. Given an instance of a class representing a syntactic element, such as the "Solid Transformation" class shown in Figure 7, the CODA model analyzer aims to convert the element to an instance of a leaf-level class representing an appropriate COBRA semantic concept. For example, the Brake object shown in Figure 6, input originally as a "Solid Transformation," is classified as a "Periodic Device Input Object" class, using four of the six classification rules shown in Figure 7. Only four rules were used in this example because in cases where alternate inheritance paths exist between superclasses and subclasses, only one of the paths is selected during the actual classification process, which is covered in Section 5.2.

The COBRA metamodel then consists of three major components: (1) a taxonomy that organizes, as a web of inheritance relationships, the syntactic elements on a flow diagram and the semantic concepts implied by the COBRA analysis method, (2) a set of defining axioms assigned to each concept, and (3) a classification rule assigned to each inheritance path between syntactic elements and leaf-level semantic concepts. These components, when coded with CLIPS knowledge representation techniques, can be used by the CODA model analyzer to assign semantic tags to syntactic elements on a flow diagram, to check that all elements on the diagram have leaf-level semantic tags from the taxonomy, and to determine whether each tagged element on a flow diagram satisfies the appropriate defining axioms. The following paragraphs describe how these functions are achieved.

5.2 Classifying Concepts

CLIPS includes an inference engine that can perform pattern matching on instances of CLIPS classes that are declared to be pattern-reactive. Since every class in the COBRA metamodel was declared pattern-reactive, in-



```
Classify Interface Object
if      the concept is a Solid Transformation and
        (the concept is a sink for a Directed Arc that has a Terminator as its source or
        the concept is a source for a Directed Arc that has a Terminator as its sink)
then    reclassify the concept as an Interface Object
fi

Classify Device Interface Object
if      the concept is an Interface Object and
        (the concept is the sink of an Input whose source is a Device or
        the concept is the source for an Output whose sink is a Device or
        the concept is the sink for an Interrupt whose source is a Device)
then    reclassify the concept as a Device Interface Object
fi

Classify Device Input Object
if      the concept is a Device Interface Object and
        the concept is the sink for an Interrupt or Input and
        the concept is not the source for an Output
then    reclassify the concept as a Device Input Object
fi

Classify Periodic Device Interface Object
if      the concept is a Device Interface Object and
        the concept is the sink for a Timer and
        the concept is not the sink for an Interrupt
then    reclassify the concept as a Periodic Device Interface
Object
fi

Classify Periodic Device Input Object Path One
if      the concept is a Device Input Object and
        the concept is the sink for a Timer and
        the concept is not the sink for an Interrupt
then    reclassify the concept as a Periodic Device Input Object
fi

Classify Periodic Device Input Object Path Two
if      the concept is a Periodic Device Interface Object and
        the concept is the sink for an Input and
        the concept is not the source for an Output
then    reclassify the concept as a Periodic Device Input Object
fi
```

Fig. 7. Relationship between taxonomy classes, axioms, and classification rules.

stances of those classes can be collected together into a facts base accessible to the CLIPS inference engine. In addition, the classification rules designed for the COBRA metamodel can be transformed into CLIPS production rules and then provided to the inference engine as a rule base. Figure 8 shows how this approach was used to implement the COBRA concept classifier within the CODA model analyzer.

First, the classification rules from the COBRA metamodel are encoded into CLIPS rule syntax and then partitioned into four rule bases. The Arc Classification partition contains 23 rules that classify most of the solid and dashed directed arcs, all terminators, and a few transformations, as shown under Phase One Classifications in the fact base portion of Figure 8. As shown at the top of Figure 8, an experienced designer will be queried regarding the identity of terminators on the flow diagram; however, the classification rules make assumptions whenever the designer cannot provide useful guidance. Three classes of concepts (Solid Transformation, Device Interface Object, and Data Flow) might prove unclassifiable by the first rule partition, and so must be deferred for a later partition.

The second partition, Transformation Classification, contains 34 rules that classify the bulk of the solid transformations on a flow diagram without any interaction with the designer. The set of concepts that can be classified by the second partition is given under Phase Two Classifications in the facts-base portion of Figure 8. The few residual elements requiring further classification after the second partition consist of ambiguous data-flow pairs and ambiguous functions, whether triggered or aperiodic.

The third partition, Stimulus-Response Classification, interacts with an experienced designer to determine which of each pair of counter-directed data flows is activated first (Stimulus), and which is sent in reply (Response). Of course, such data flows can also be completely independent.

The fourth partition interacts with an experienced designer to resolve any ambiguities remaining when attempting to classify Solid Transformations as functions. In some cases, the classification rules cannot determine whether a function operates independently or under direction of another function. Information provided by the designer can help to resolve such ambiguities. For example, the designer might know something about the time taken to perform a function, or might know whether another function must suspend pending input from a connected function. Whenever the designer is not experienced or cannot provide help, the classification rules make default assumptions.

Two other issues are worth noting. First, the taxonomy was designed so that each rule is derived from one inheritance path. Because the taxonomy includes multiple inheritance, more than one path may be active simultaneously; however, once a rule fires choosing one of the paths, the competing rules on the unchosen paths will no longer be active because their patterns no longer match the facts. Second, while in most cases the objects input to the facts base correspond to syntactic elements on a flow diagram, the rules are devised so that input objects may also be partially classified by the designer. This allows a designer to work with semantic concepts at mixed

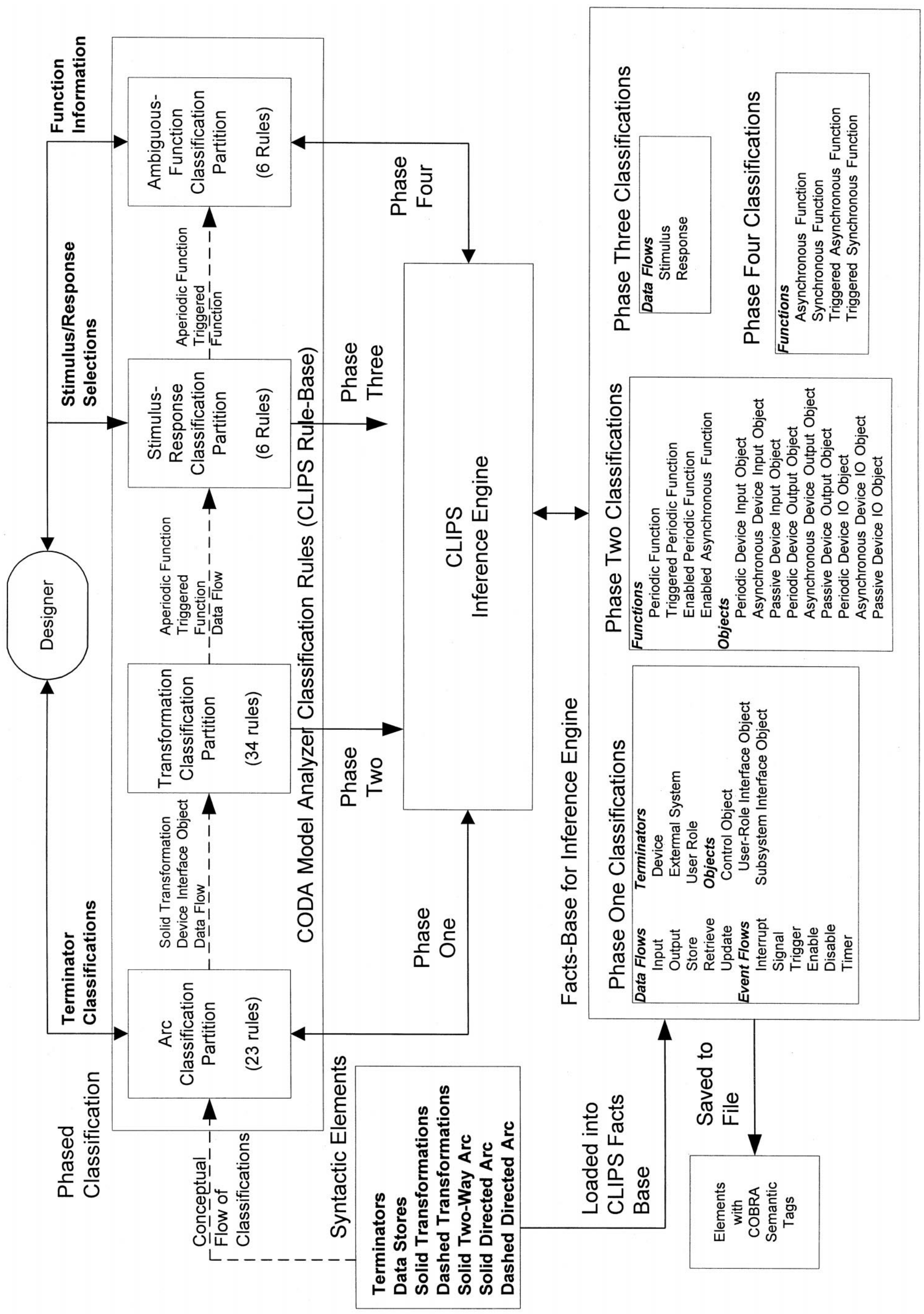

Phased Classification
Terminator Classifications
Designer
Stimulus/Response Selections
Function Information
CODA Model Analyzer Classification Rules (CLIPS Rule-Base)
Arc Classification Partition (23 rules)
Transformation Classification Partition (34 rules)
Stimulus-Response Classification Partition (6 Rules)
Ambiguous-Function Classification Partition (6 Rules)
Solid Transformation
Device Interface Object
Data Flow
Aperiodic Function
Triggered Function
Data Flow
Aperiodic Function
Triggered Function
Conceptual Flow of Classifications
Syntactic Elements
Terminators
Data Stores
Solid Transformations
Dashed Transformations
Solid Two-Way Arc
Solid Directed Arc
Dashed Directed Arc
Phase One
Phase Two
Phase Three
Phase Four
CLIPS Inference Engine
Loaded into CLIPS Facts Base
Saved to File
Elements with COBRA Semantic Tags
Facts-Base for Inference Engine
Phase One Classifications
Data Flows
Input
Output
Store
Retrieve
Update
Event Flows
Interrupt
Signal
Trigger
Enable
Disable
Timer
Terminators
Device
External System
User Role
Objects
Control Object
User-Role Interface Object
Subsystem Interface Object
Phase Two Classifications
Functions
Periodic Function
Triggered Periodic Function
Enabled Periodic Function
Enabled Asynchronous Function
Objects
Periodic Device Input Object
Asynchronous Device Input Object
Passive Device Input Object
Periodic Device Output Object
Asynchronous Device Output Object
Passive Device Output Object
Periodic Device IO Object
Asynchronous Device IO Object
Passive Device IO Object
Phase Three Classifications
Data Flows
Stimulus
Response
Phase Four Classifications
Functions
Asynchronous Function
Synchronous Function
Triggered Asynchronous Function
Triggered Synchronous Function

Fig. 8. CLIPS implementation of the concept classifier within the model analyzer.

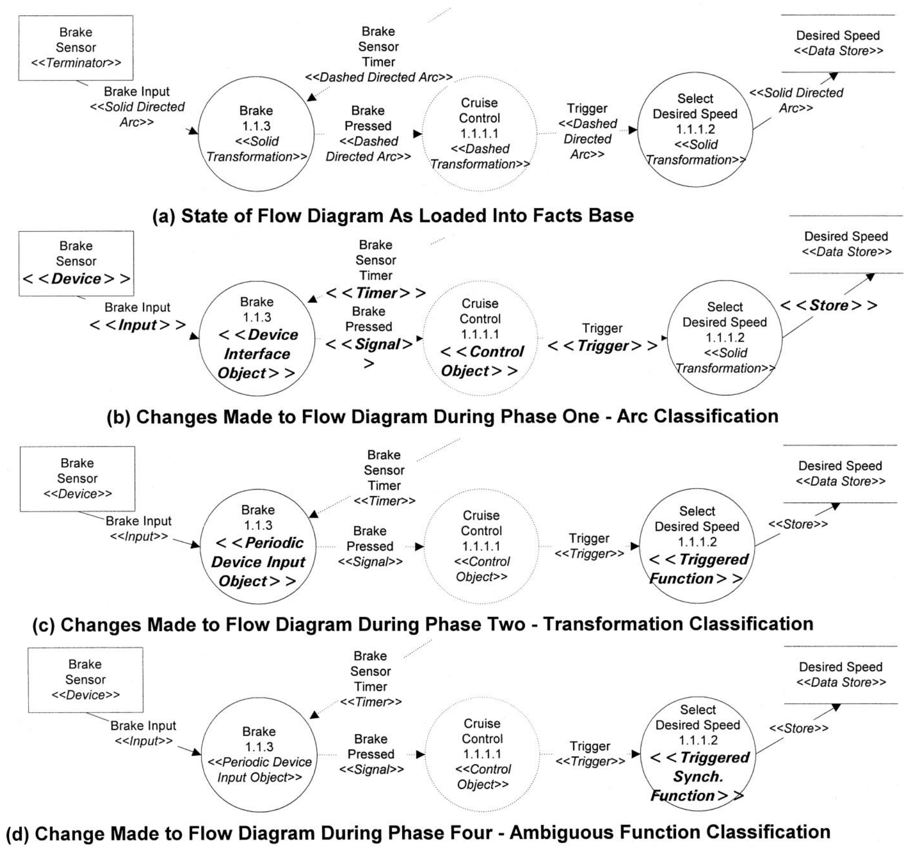

Brake Sensor <<Terminator>>
Brake Input <<Solid Directed Arc>>
Brake 1.1.3 <<Solid Transformation>>
Brake Sensor Timer <<Dashed Directed Arc>>
Brake Pressed <<Dashed Directed Arc>>
Cruise Control 1.1.1.1 <<Dashed Transformation>>
Trigger <<Dashed Directed Arc>>
Select Desired Speed 1.1.1.2 <<Solid Transformation>>
<<Solid Directed Arc>>
Desired Speed <<Data Store>>
(a) State of Flow Diagram As Loaded Into Facts Base
Brake Sensor <<Device>>
Brake Input <<Input>>
Brake 1.1.3 <<Device Interface Object>>
Brake Sensor Timer <<Timer>>
Brake Pressed <<Signal>>
Cruise Control 1.1.1.1 <<Control Object>>
Trigger <<Trigger>>
Select Desired Speed 1.1.1.2 <<Solid Transformation>>
<<Store>>
Desired Speed <<Data Store>>
(b) Changes Made to Flow Diagram During Phase One - Arc Classification
Brake Sensor <<Device>>
Brake Input <<Input>>
Brake 1.1.3 <<Periodic Device Input Object>>
Brake Sensor Timer <<Timer>>
Brake Pressed <<Signal>>
Cruise Control 1.1.1.1 <<Control Object>>
Trigger <<Trigger>>
Select Desired Speed 1.1.1.2 <<Triggered Function>>
<<Store>>
Desired Speed <<Data Store>>
(c) Changes Made to Flow Diagram During Phase Two - Transformation Classification
Brake Sensor <<Device>>
Brake Input <<Input>>
Brake 1.1.3 <<Periodic Device Input Object>>
Brake Sensor Timer <<Timer>>
Brake Pressed <<Signal>>
Cruise Control 1.1.1.1 <<Control Object>>
Trigger <<Trigger>>
Select Desired Speed 1.1.2 <<Triggered Synch. Function>>
<<Store>>
Desired Speed <<Data Store>>
(d) Change Made to Flow Diagram During Phase Four - Ambiguous Function Classification

Fig. 9. Monitoring changes to a flow-diagram fragment during concept classification.

levels of abstraction, and yet the model analyzer can still classify the concepts more specifically.

As Figure 8 shows, CLIPS objects representing the syntactic elements of a flow diagram (e.g., as given earlier in Figure 6) are loaded into a CLIPS facts base, and the inference engine is invoked in four phases, one for each partition of the rule base. At the close of each phase, the facts base has been updated by the inference engine, based on actions from the rules that fired during the phase. Figure 9(a) gives a visual representation of some CLIPS objects loaded into the facts base. The objects shown in Figure 9(a) correspond to the flow-diagram fragment shown earlier in Figure 6, but the classification of each object is denoted with semantic tags enclosed in guillemets, ⟨⟨Solid Transformation⟩⟩ for example, rather than with the CLIPS syntax shown in Figure 6. Figure 9(b) shows the changes made to the input flow-diagram fragment as a result of executing the first classification phase. Notice that concept classification has moved from the syntactic to the semantic for 9 of the 10 objects. The data store object, Desired Speed, was not reclassified because the input notation includes a symbol for data store, which is a leaf-level concept in the COBRA taxonomy. At the

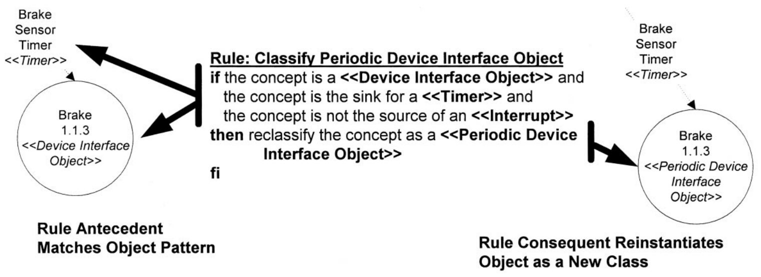

Brake Sensor Timer <<Timer>>
Brake 1.1.3 <<Device Interface Object>>
Rule: Classify Periodic Device Interface Object
if the concept is a <<Device Interface Object>> and
the concept is the sink for a <<Timer>> and
the concept is not the source of an <<Interrupt>>
then reclassify the concept as a <<Periodic Device Interface Object>>
fi
Brake Sensor Timer <<Timer>>
Brake 1.1.3 <<Periodic Device Interface Object>>
Rule Antecedent Matches Object Pattern
Rule Consequent Reinstantiates Object as a New Class

Fig. 10. Example COBRA metamodel classification rule.

end of this first phase of classification all but two objects have been reclassified as leaf-level concepts in the taxonomy. The Brake object has been partially classified, as a ⟨⟨Device Interface Object⟩⟩, while the Select Desired Speed object remains at the syntactic level, ⟨⟨Solid Transformation⟩⟩. Figure 9(c) shows the two changes made to the flow-diagram fragment as a result of rule executions during phase two. The Brake object has now been classified as a leaf-level concept, ⟨⟨Periodic Device Input Object⟩⟩, while the Select Desired Speed object has been further classified as a ⟨⟨Triggered Function⟩⟩. The flow diagram does not change during the third classification phase. As shown in Figure 9(d), a final classification occurs during the fourth phase of classification. In this case, the Select Desired Speed object is classified as ⟨⟨Triggered Synchronous Function⟩⟩, a leaf-level concept in the taxonomy. To summarize, a CLIPS object that enters the classification process as a syntactic element will typically undergo several reclassifications before finally being classified as a leaf-level concept in the COBRA concept taxonomy, described earlier in Section 5.1. The total number of rule firings required to classify an entire flow diagram depends upon the number and type of elements in the diagram and on the degree of classification present prior to invoking the concept classifier.

To understand in detail how classification rules work, we consider now one specific rule, the rule that reclassifies a ⟨⟨Device Interface Object⟩⟩ as a ⟨⟨Periodic Device Interface Object⟩⟩. Note that this rule is contained in the second partition of the concept classification rules. Figure 10 provides a conceptual view of the rule. The rule antecedent detects a pattern relating a node, ⟨⟨Device Interface Object⟩⟩, to an arc, ⟨⟨Timer⟩⟩, and detects the absence of an interfering arc, ⟨⟨Interrupt⟩⟩. The rule consequent then reinstantiates the node with a new classification, ⟨⟨Periodic Device Interface Object⟩⟩, records the fact that the classification rule fired, and then deletes the old object because the new object has subsumed all information formerly held by the old object. Each rule partition contains a set of rules that appear similar to the one discussed here. While executing a given rule partition, the CLIPS inference engine simply activates all rules with antecedents that match object patterns in the facts base, selects one activated rule for execution, and then executes the selected rule. This cycle continues until no rules in the partition have antecedents that match object patterns in the facts base.

Once all rule partitions have been executed, each CLIPS object in the facts base should be classified as one of the leaf-level semantic concepts in the taxonomy. To ensure this, each object inherits an operation, check-classification, from the Model Element class. When invoked, the operation determines whether or not its own instance can be used to create a subclass. When classification is complete, then no object in the flow diagram should be capable of being "subclassed."

5.3 Eliciting Information

Once the flow diagram is classified, the designer might need to specify some information about input rates and timer periodicity in order to guide design decisions made by the design generator. Immediately following concept classification, the model analyzer elicits the needed information automatically from the designer. For every type of flow-diagram element that requires additional information, the model analyzer can determine if the information has been supplied and, if not, can prompt the designer for the information. To implement this process, the model analyzer uses the CLIPS object-oriented query language. For example, a CLIPS function, **elicit-timer-periods**, searches the facts base for all objects of type "Timer" that have no period assigned. For each such object, the designer is prompted with the identity of the object and is asked to provide a positive value for the period attribute. Once a proper period is provided, the object is updated appropriately, and the elicitation is logged in the design record. After all needed information has been supplied for all appropriate elements in the flow diagram, then the facts base should be ready for immediate input to the CODA design generator. The facts base can be saved to a file for later use by the design generator; however, to ensure that the flow diagram is well formed, the designer must verify that all axioms are satisfied for each object on the diagram. The following paragraphs describe the implementation of axiom checking for the model analyzer.

5.4 Checking Axioms

Recall from Figure 7 that each class in the COBRA metamodel taxonomy can be augmented with axioms that must be satisfied. Any such axioms that apply to a class are encapsulated as private operations within a public operation, **check-axioms**. For example, consider the class "Periodic Device Interface Object," which must satisfy two axioms: "One, And Only One, Timer" and "No Interrupt." Each axiom is specified using the CLIPS object-oriented query language. The first axiom works in two steps. First, the facts base is searched for all instances of type "Timer," where the sink of the instance corresponds to the object being evaluated. Second, the resulting set is tested to ensure it contains only one member. If not, then an axiom violation is reported. The second axiom works in a single step by searching to find any instance of type "Interrupt," where the sink of the instance corresponds to the object being evaluated. If any such instance is found, then an axiom violation is reported.

To check whether or not a flow diagram satisfies all axioms, the designer simply invokes a **check-axioms** function from the CODA user interface. The function loops through each object that composes the flow diagram and calls the **check-axioms** operation. The CLIPS object-oriented run-time system provides the execution algorithm for checking every axiom embedded in each class within the inheritance path of an object. The algorithm works as follows.

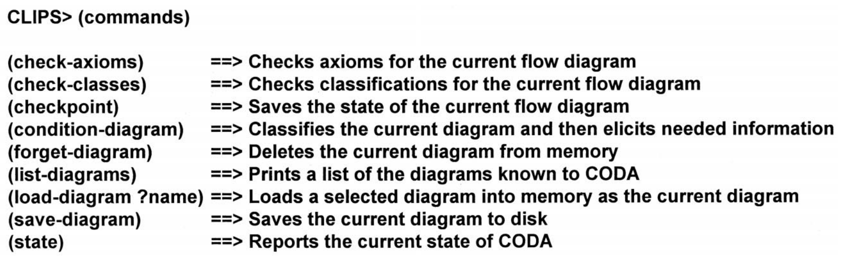
The top-level class in the taxonomy, "Model Element," contains an empty *primary* method that implements **check-axioms**. When the **check-axioms** operation is called on any object, it is this *primary* method that is invoked. Every other subclass of the class "Model Element" contains an *after* method also called **check-axioms**. Any relevant axiom-checking queries are embedded within these *after* methods. Once the *primary* method, **check-axioms**, completes, the CLIPS run-time works its way down the inheritance hierarchy for an object. In each subclass of "Model Element" the *after* method called **check-axioms** is executed. Within each *after* method, every CLIPS query is performed in turn. In this manner, once the final **check-axioms** *after* method completes execution for the lowest-level class that composes an object, all the axioms pertaining to an object are evaluated, and the designer is notified of any violations. Of course, if the concept classifier was invoked successfully, then all axioms will be satisfied, and no violations will be reported.

5.5 User View of the Model Analyzer

While the previous discussion covered internal design and implementation, the interaction between the user and the model analyzer remains to be discussed. Two modes of interaction are supported. For the self-declared inexperienced designer, CODA leads the designer step-by-step through the process of loading the objects corresponding to a flow diagram, classifying the concepts and providing additional information, and checking for proper classification and to ensure that all axioms are satisfied. Alternatively, an experienced designer is presented with a set of commands that can be invoked individually whenever the designer wishes. For each command that can potentially alter the flow diagram, CODA checks that relevant constraints are satisfied when a designer invokes the command. For example, if a designer attempts to generate a design before a flow diagram has been properly classified, then CODA will remind the designer that more preparation is needed. Figure 11 gives the commands that the model analyzer makes available to an experienced designer. The design generator provides additional commands.

6. USING CODA TO DESIGN A CRUISE-CONTROL SYSTEM

The CODA model analyzer was applied to the flow diagram for an automobile cruise-control and monitoring system. The entire flow diagram for this



```
CLIPS> (commands)

(check-axioms)        ==> Checks axioms for the current flow diagram
(check-classes)       ==> Checks classifications for the current flow diagram
(checkpoint)          ==> Saves the state of the current flow diagram
(condition-diagram)   ==> Classifies the current diagram and then elicits needed information
(forget-diagram)      ==> Deletes the current diagram from memory
(list-diagrams)       ==> Prints a list of the diagrams known to CODA
(load-diagram ?name) ==> Loads a selected diagram into memory as the current diagram
(save-diagram)        ==> Saves the current diagram to disk
(state)               ==> Reports the current state of CODA
```

Fig. 11. CODA model analyzer commands accessible to an experienced designer.

system consists of 58 nodes (33 transformations, 12 terminators, and 13 data stores) and 112 arcs (69 data flows and 43 event flows). Figure 12 shows only a fragment taken from the larger flow diagram. Two data flows, labeled Current Speed, come from a data store omitted from the fragment, and the four event flows classified ⟨⟨Timer⟩⟩ come from a system terminator also omitted from the fragment. The flow-diagram fragment is annotated with information inferred by the CODA model analyzer or elicited from the designer. The annotations, set off in italicized print, are shown in two forms: guillemets enclose semantic tags, and square brackets enclose supplementary information. Symbols, defined in Table II, indicate the source of the annotations. The example in Figure 12 uses all symbols except for the ? symbol.

6.1 Analyzing the Flow Diagram Model

Classifying concepts for this flow diagram requires a dialog between CODA and the designer; however, CODA can make most of the classification decisions without consulting the designer. At startup, CODA prompts for the designer's level of experience. When the designer is inexperienced CODA makes all decisions without consulting the designer. When the designer is experienced, as in this case study, CODA consults the designer from time to time, where such consultation might prove advantageous. In this case, only two consultations were used. First, the designer, when asked about the nature of the terminators in the model, indicates that all terminators are devices. Second, during the latter stages of classification, CODA discovers six data transformations (none included in Figure 12) that appear to be synchronous functions. Knowing the designer to be experienced, CODA presents each of these tentative classifications to the designer for confirmation.

After completing concept classification, CODA notes that some of the new semantic tags assigned to symbols on the flow diagram require the designer to supply additional information. For example, in this case study, 16 event flows represent timers (four shown in Figure 12). CODA enables the designer to provide a positive period for each timer. After finishing concept classification and information elicitation, CODA can verify that each symbol on the flow diagram is properly tagged and that each tagged symbol satisfies its definition in the COBRA metamodel. A full treatment of this

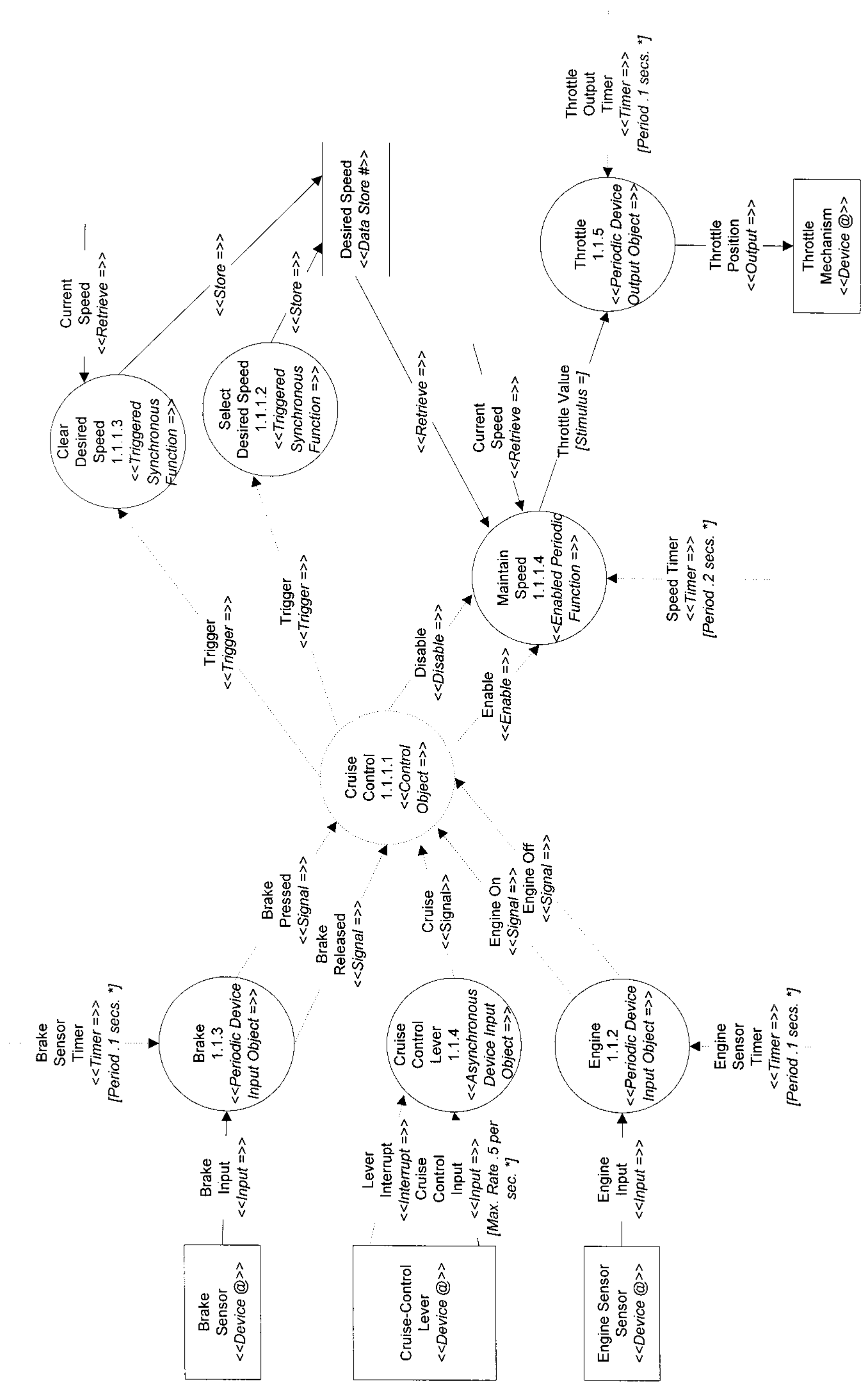

Brake Sensor <<Device @>>
Brake Input <<Input =>>
Brake 1.1.3 <<Periodic Device Input Object =>>
Brake Sensor Timer <<Timer =>> [Period .1 secs. *]
Cruise-Control Lever <<Device @>>
Lever Interrupt <<Interrupt =>>
Cruise Control Input <<Input =>> [Max. Rate .5 per sec. *]
Cruise Control Lever 1.1.4 <<Asynchronous Device Input Object =>>
Engine Sensor Sensor <<Device @>>
Engine Input <<Input =>>
Engine 1.1.2 <<Periodic Device Input Object =>>
Engine Sensor Timer <<Timer =>> [Period .1 secs. *]
Brake Pressed <<Signal =>>
Brake Released <<Signal =>>
Cruise <<Signal>>
Engine On <<Signal =>>
Engine Off <<Signal =>>
Cruise Control 1.1.1.1 <<Control Object =>>
Trigger <<Trigger =>>
Trigger <<Trigger =>>
Disable <<Disable =>>
Enable <<Enable =>>
Clear Desired Speed 1.1.1.3 <<Triggered Synchronous Function =>>
Current Speed <<Retrieve =>>
Select Desired Speed 1.1.1.2 <<Triggered Synchronous Function =>>
<<Store =>>
<<Store =>>
Desired Speed <<Data Store #>>
<<Retrieve =>>
Current Speed <<Retrieve =>>
Maintain Speed 1.1.1.4 <<Enabled Periodic Function =>>
Speed Timer <<Timer =>> [Period .2 secs. *]
Throttle Value [Stimulus =]
Throttle 1.1.5 <<Periodic Device Output Object =>>
Throttle Output Timer <<Timer =>> [Period .1 secs. *]
Throttle Position <<Output =>>
Throttle Mechanism <<Device @>>
Fig. 12. Flow-diagram fragment from a COBRA model of a cruise control system.

Table II. Symbols Used to Annotate Flow Diagrams

Symbol	Meaning
#	This classification is directly represented using a visual symbol.
=	CODA inferred this classification.
?	CODA tentatively inferred this classification, but the designer was asked to confirm or override the classification.
+	CODA elicited additional information from the designer and then inferred this classification based on that additional information.
@	CODA elicited this classification from the designer.
*	CODA elicited this additional information from the designer after classifying the concept and determining that some required information was missing.

case study, including a complete, annotated flow diagram and multiple concurrent designs, appears elsewhere [Mills 1996]. To keep this case study brief, only the fragment in Figure 12 is shown here.

6.2 Generating a Concurrent Design

The model analyzer provides a front-end to a design generator that encodes heuristics from a specific design method, Concurrent Design Approach to Real-Time Systems (CODARTS) [Gomaa 1993]. The design generator transforms a COBRA flow diagram into a concurrent design by implementing the four steps used in CODARTS: (1) task structuring, (2) task interface definition, (3) module structuring, and (4) task and module integration. Figure 13 provides an overview of the design generated by CODA from the flow-diagram fragment shown in Figure 12. Please note that some portions of the design correspond to symbols omitted from Figure 12. Specifically, the task, Control Auto Speed, encapsulates transformations that maintain, increase, and resume speed, while Figure 12 shows only the speed maintenance transformation. Similarly, while the Current Speed data store is omitted from Figure 12, Figure 13 contains a module that corresponds to the data store. Several examples will serve to illustrate how the CODA design generator depends upon the semantic tags added by the model analyzer.

As an example of task structuring, CODA creates a single periodic input task, Monitor Auto Sensors, to poll both the Brake and Engine objects from Figure 12, because both transformations are tagged ⟨⟨Periodic Device Input Object⟩⟩ and their associated ⟨⟨Timer⟩⟩ event flows have identical periods, [Period .1 secs]. As an example of task interface definition, CODA creates a message queue, Cruise Control Events, to hold asynchronous events flows, tagged ⟨⟨Signal⟩⟩, sent from several other tasks to the Control Cruising task. A message queue is chosen because the destination task contains the receiving transformation Cruise Control, tagged ⟨⟨Control Object⟩⟩, which hides an embedded finite-state machine that must not miss any events and that requires the order of the arriving events to be preserved. As an example of module structuring, CODA forms a data abstraction module, Desired Speed, with three operations: Select, Clear, and Read. CODA

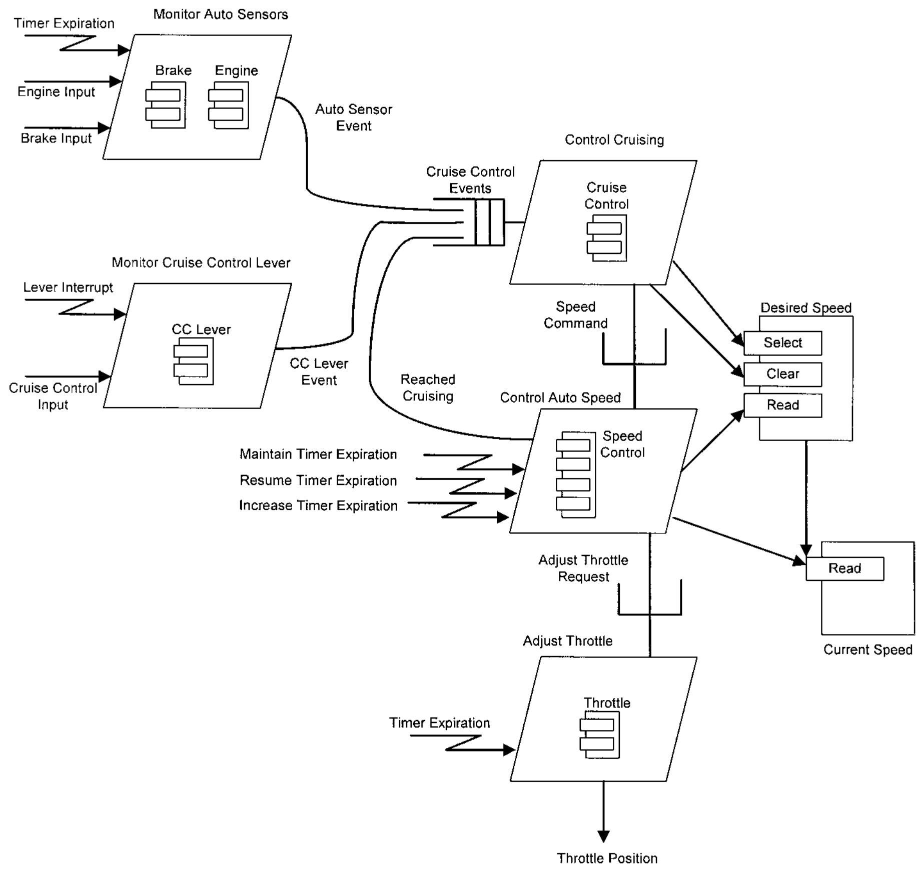


Fig. 13. CODARTS design generated from the flow diagram in Figure 12.

generates this module from six diagram symbols shown in Figure 12: one ⟨⟨Data Store⟩⟩, Desired Speed, two transformations, Clear Desired Speed and Select Desired Speed, tagged ⟨⟨Triggered Synchronous Function⟩⟩ and two data flows tagged ⟨⟨Store⟩⟩ and one tagged ⟨⟨Retrieve⟩⟩. As an example of task and module integration, CODA places the Desired Speed module outside any task, because the module is shared by two tasks, Control Cruising and Control Auto Speed. In addition, CODA designates that the Select and Clear operations are invoked by Control Cruising, while the Read operation is invoked by Control Auto Speed. CODA makes these decisions by simultaneously examining the data flow model, the evolving concurrent design, and the relationships among elements on both. Details describing the internal operation of the CODA design generator appear elsewhere [Mills 1996; Mills and Gomaa 1996].

7. EVALUATION

The CODA model analyzer for COBRA was applied to four real-time systems modeled using the visual notation shown earlier in Figure 2. In

Table III. Classifications Over All Models for the Case Studies

	Total	Directly Represented	Inferred Automatically	Inferred Tentatively	Required Information	Designer Classified
All Nodes	125	18 (14%)	59 (47%)	8 (6%)	12 (10%)	28(22%)
Transformations	79	0 (0%)	59 (75%)	8 (10%)	12 (15%)	0 (0%)
Terminators	28	0 (0%)	0 (0%)	0 (0%)	0 (0%)	28 (100%)
Data Stores	18	18 (100%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
All Arcs	233	0 (0%)	231 (99%)	0 (0%)	2 (1%)	0 (0%)
Event Flows	91	0 (0%)	91 (100%)	0 (0%)	0 (0%)	0 (0%)
Data Flows	142	0 (0%)	140 (99%)	0 (0%)	2 (1%)	0 (0%)
All Elements	358	18 (5%)	290 (81%)	8 (2%)	14 (4%)	28 (8%)

addition to the Automobile Cruise-Control and Monitoring System, just discussed in the preceding section, these systems included a robot controller [Gomaa 1993], an elevator control system [Gomaa 1993], and a remote temperature sensor [Nielsen and Shumate 1987; 1988].

7.1 Summary of Results

Table III presents an overview of the classification results across all case studies. For conciseness, the syntactic elements from Figure 2 are represented in five rows within Table III. Row two, Transformations, includes both data and control transformations; row seven, Data Flows, includes both unidirectional and bidirectional data flows. Row one of Table III represents all nodes on all flow diagrams; row five represents all arcs; and row eight represents all elements (that is, nodes plus arcs). The first column in Table III simply gives the total number of each type of element that appears on the flow diagrams in the case studies; specifically, the case-study diagrams contained 358 elements as follows: (1) 125 nodes of which 79 were transformations, 28 were terminators, and 18 were data stores and (2) 233 arcs of which 91 were event flows and 142 were data flows. The remaining five columns represent the manner in which classification is achieved for the elements represented by each intersecting row. A review of the last row indicates the degree of success achieved when the CODA model analyzer is employed against the flow diagrams for the four case studies. As shown in columns Directly Represented and Inferred Automatically, CODA succeeded without help in classifying about 86% of the elements in the flow diagrams. Where no human assistance is available, CODA makes default decisions that would have proven accurate in all but one of the 358 classification decisions comprising the four case studies. CODA achieves this success by invoking default rules that take reasonable classification decisions in the absence of additional guidance. This ability to make effective classifications without human intervention allows the CODA design generator to be used even though human assistance is unavailable or unwanted. For the case studies reported in this paper, human assistance was available and provided.

As shown in the last column of Table III, the designer provided the classification for terminators, 8% of the elements in the case studies. Cases

Table IV. Classification of Transformations by Case Study

Case Study	Transformations	Inferred Automatically	Inferred Tentatively	Required Information
All Models	79	59 (75%)	8 (10%)	12 (15%)
Automobile Cruise Control	33	27 (82%)	6 (18%)	0 (0%)
Robot Controller	18	14 (78%)	1 (5%)	3 (17%)
Elevator Control System	14	13 (93%)	0 (0%)	1 (7%)
Remote Temperature Sensor	14	5 (36%)	1 (7%)	8 (57%)

worth considering in detail involve those 2% where CODA's classification was inferred tentatively but referred to the designer for confirmation and those 4% where CODA's classification occurred only after the designer supplied additional information. In these cases, a large difference exists between the classification of arcs, where 99% are classified without help, and the classification of transformations, where only 75% are classified without help.

7.2 Problems Classifying Data Flows

Only two data flows, from among the 233 arcs considered in the case studies, were classified with help from the designer. Both of these appear in the same flow diagram; in fact, the two data flows are related. In this situation two data transformations exchange data flows with each other. CODA cannot determine which of these two data flows, if either, is sent in response to the other; therefore, the designer must be consulted. The designer should know, or be able to determine from the pseudocode associated with each data transformation, whether one of the data flows is sent in response to the other. Of course, the designer might not know this information, so CODA is prepared to make a default decision. Only in situations such as this one will CODA's automated classifier need to consult the designer to classify arcs on a flow diagram. Thus, the performance of the automated classifier against arcs will depend on the number of these cases that exist in a given diagram. The performance of CODA's automated classifier against transformations appears less effective.

7.3 Problems Classifying Transformations

Table IV provides a breakdown of transformation classifications by case study. For the four case studies, CODA could classify definitively 75% of the transformations, could classify tentatively 10%, and could classify the remaining 15% only after hints from the designer. All control objects are inferred automatically from their syntax.

CODA's performance is much better in three of the case studies: automobile cruise control, robot controller, and elevator control system. Considering only these three cases, 83% (54/65) of transformations were classified automatically; 11% (7/65) could be classified tentatively, while only 6% (4/65) required hints in order to make a classification. For the remote temperature sensor, CODA's classification performance proved less effective. The data flow diagram for the remote temperature sensor, as provided by Nielsen and Shumate [1987; 1988], was developed using structured analysis as the design method. As applied in this case, the design method did not include semantic concepts for interpreting the flow diagrams. Even though the analysis model for the remote temperature sensor was developed using only a subset of the COBRA syntax, and without using the semantic concepts included in COBRA, the COBRA model analyzer, working together with the designer, was still able to produce a semantic classification of the syntactic elements used in the flow diagram. Even without interacting with the designer, and thus using only its default classification rules, the CODA model analyzer would have reached the same classification decisions for all but one of the elements included on the flow diagram for the remote temperature sensor application. The following paragraphs discuss and analyze those situations where CODA's model analyzer for COBRA requested the designer to confirm tentative classifications or to provide assistance with classification.

7.3.1 *Tentative Classifications.* In the case studies, CODA tentatively classified eight transformations, each involving the same type of situation. Whenever CODA encounters a function on a flow diagram such that the function sends data only to data stores or passive device-interface objects, or to both, then, if the function is not classified otherwise, CODA tentatively identifies the function as synchronous. This tentative classification assumes that, for real-time systems, updating data stores and writing to passive devices is generally a fast operation that can be completed atomically. This assumption is usually correct. In some situations, however, an operation might take long enough that the designer chooses to view the function as asynchronous. In the particular case studies covered in this paper for example, the designer confirms the tentative classification in seven of the eight cases. In one case, occurring within the remote temperature sensor application, the designer overrides CODA's tentative classification where a function updates a data store. The designer overrides CODA based on application-specific knowledge that the data store is large enough, or that the update algorithm is time-consuming enough, to warrant asynchronous processing. This information is not available to CODA but might be available to the designer. When the designer does not know whether to override CODA's tentative decision, then the decision stands.

7.3.2 *Assisted Classifications.* In some situations CODA recognizes that additional information might be available that can help make a more accurate classification. In these situations, represented in the last column of Table IV, CODA consults the designer to see what other information exists. Lacking additional information, CODA makes a default classification. Table IV indicates 12 instances among the case studies where the designer is consulted to help classify a transformation. These instances represent two general situations. The first situation occurs when a function is triggered by a control transformation, yet the triggered function receives input data from some other transformation. In situations of this type, CODA recognizes that the triggered function might not be able to execute at the triggering state-transition because the input data might be unavailable. The designer is consulted on this question, and CODA then makes the best classification based upon any additional information provided. The second, and more frequent, situation occurs when a function receives input from only a single source, or the same input from multiple sources. In such situations, CODA recognizes that the function might be classified as either a synchronous or asynchronous function depending upon the execution time required. After consulting the designer, CODA makes the best classification based on the information available. Had no human assistance been available, CODA's default decisions would have proven accurate in all but one of the 79 transformation classifications reported in Table IV.

7.4 Automated Design Generation

In large part, the effectiveness of CODA's model analyzer can be evaluated based on the results produced by CODA's design generator. The design generator was used to automatically generate 10, distinct, concurrent designs for the preceding four real-time systems, modeled as flow diagrams [Mills 1996]. Multiple designs were generated for each system to test the ability of CODA's design generator to adapt to variations in the intended target environment. Of the 1,568 CODARTS design decisions required to generate the 10 designs, 1,524, or 97%, were taken without human assistance. The effectiveness of CODA's design generator derives in large measure from CODA's model analyzer, which creates the semantic view from COBRA needed to effectively apply the CODARTS heuristics encoded within CODA's design generator.

8. DISCUSSION

The approach described in this paper can be applied to assist designers in the creation of concurrent designs. A tool such as CODA could be embedded as a component in a computer-aided software engineering (CASE) tool. Most CASE tools enable a designer to enter flow diagrams and structure charts, or other visual models of a software design; however, the process of creating the software design from the flow diagrams must be performed by a human designer, outside the CASE tool and without automated assistance. Where a component such as CODA is available, a designer could enter a flow diagram into a CASE tool and then invoke automated assistance to generate a design. Such automation can capture design decisions and rationale and can maintain traceability between elements on the flow diagram and components in the design. Beyond assisting designers, a tool like CODA can also be used to train novice software designers. For example, novice designers could compare their models and designs to the results produced by CODA for the same problems. In addition, since CODA captures design rationale, including a detailed history of the design decisions made for every element in an analysis model and a design, novice designers could study the decisions made by CODA as an aid to learn analysis and design methods that have been automated within CODA.

While the knowledge-based approach embodied in CODA was used to automate the transformation of COBRA visual models into CODARTS designs, the approach should be applicable to a range of software design methods that model system behavior using visual notations and accompanying textual descriptions. The results obtained for COBRA indicate that the classification by CODA proves more effective when the model analyzer encounters models constructed using the semantic concepts provided by COBRA; however, acceptable results were still obtained when the COBRA model analyzer was applied to a model developed using structured analysis, which does not use the COBRA semantics. This indicates that the approach described in this paper works better with an analysis method that provides semantic concepts, where each transformation has a specific role in the model, over a method that only provides syntactic concepts. CODA assumes the existence of semantic concepts in order to classify syntactic symbols on a diagram.

Increasingly, software analysis and design methods that use visual notations are also including an underlying semantic model that can be exploited using the approach outlined in this paper. For example, consider the Unified Modeling Language (UML), which includes an underlying metamodel to provide some semantic foundation for its visual models [Fowler 1997; Booch et al. 1999; Jacobson et al. 1999; Rumbaugh et al. 1999]. The UML visual models include a number of diagrams: class diagrams, collaboration diagrams, sequence diagrams, statecharts [Harel 1988; Harel and Gary 1996], activity diagrams, component diagrams, deployment diagrams, and package diagrams. UML defines each diagram visually with symbols from an underlying UML metamodel. Aside from providing some semantics for each diagram, the UML metamodel provides a primary means of extending the UML through stereotypes. Stereotypes can be used as a tag assigned to UML elements that can be extended. For example, a UML Actor is actually a UML Class with an assigned stereotype, ⟨⟨Actor⟩⟩. By arranging stereotypes in an inheritance hierarchy and by permitting stereotypes to exhibit relationships with other stereotypes and UML elements, a semantic metamodel emerges for UML and for extensions to UML. Oddly, for the basic diagrams defined by UML, the metamodel exhibits one major omission: few semantic relationships are defined among the various diagrams. Overcoming this omission would increase the power of UML semantics.

The semantic concepts discussed in this paper appear quite similar in intent to stereotypes within the Unified Modeling Language (UML). A taxonomy of UML stereotypes, including defining axioms, can be devised in a form similar to that depicted in this paper for the COBRA concept taxonomy. Rules could then be formulated for attempting to classify UML model elements against the taxonomy of stereotypes. In addition, UML model elements labeled with stereotypes by a designer could be evaluated automatically against a relevant hierarchy of defining axioms. Model elements with associated stereotypes could also trigger the automatic elicitation of information required to support later phases of the software design process. Once a UML model is properly labeled with stereotypes and augmented with additional information, the CODARTS design heuristics encoded within CODA's design generator could be used to produce a concurrent design from a UML model.

One UML-based analysis and design method that provides a designer with semantic concepts is known as COMET, the Concurrent Object Modeling and Architectural Design Method [Gomaa 2000]. The object-structuring criteria provided by COMET are an extension of those provided by COBRA, as are the classification of external classes (terminators in COBRA). The COMET analysis method uses UML stereotypes to capture semantic concepts, such as ⟨⟨device interface⟩⟩ object or ⟨⟨state dependent control⟩⟩ object. The similarities between the COMET method and COBRA suggest that the approach should also work with UML-based design methods in which the UML notation is supplemented with semantic concepts that exploit the stereotype concept included within the UML metamodel. While providing an underlying foundation for the semantics of UML diagrams, today the UML metamodel goes largely unexploited because few automated tools have been built to check instances of UML models against the metamodel. Such checking could determine when human analysts and designers have erred during stereotype assignment. In fact, this serves to illustrate the original motivation for the work reported in this paper.

9. CONCLUSIONS

In this paper we described a knowledge-based approach to infer the presence of semantic concepts from visual models of system behavior. The approach represents knowledge about an analysis and modeling method in the form of a metamodel, consisting of a concept taxonomy and supporting classification rules and axioms. To illustrate the approach, the paper described a metamodel for a specific modeling method, COBRA, and then showed how that metamodel was implemented with CLIPS to provide a model analyzer embedded within CODA, a concurrent designer's assistant. The paper also described how a design generator embedded within CODA can use the results from the model analyzer to transform a flow diagram into a software architecture for concurrent systems. The approach appears applicable to other visual modeling methods, such as UML, used widely by software designers.

The approach was evaluated by applying CODA to analyze data flow/control flow models for four real-time systems. For the four systems, CODA inferred, automatically and correctly, the existence of 86% of all semantic concepts within the flow diagrams. Varying degrees of human assistance were used to correctly identify the remaining semantic concepts within the diagrams: in two percent of the cases CODA reached tentative classifications that a designer was asked to confirm or override; in four percent of the cases a designer was asked to provide additional information; in the remaining eight percent of the cases, all involving terminators, the designer was asked to identify the semantic concept. When providing assistance, a human designer consults the textual description accompanying the visual model for each system. Using the semantic interpretation provided by the CODA model analyzer, the CODA design generator proved very effective. During the generation of 10 distinct designs for the four case studies, the CODA design generator made 97% of all design decisions without consultation.

The CODA approach can be used to assist in generating software designs, and as a tool to help inexperienced designers understand the decisions that need to be made when designing a real-time system. Although demonstrated here within a real-time domain, the approach can be applied to other software design domains, e.g., electronic commerce. To allow this, the analysis method for the domain must explicitly provide semantic concepts used to interpret the analysis model, and to label the visual modeling notation. The design method must include rules or heuristics, which can be codified, for mapping from an analysis model to a design model.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support provided by the National Institute of Standards and Technology (NIST), and most particularly James H. Burrows. Without his support, the work reported in this paper could not have been carried out. In addition, we appreciate the insightful suggestions from Brett Strausser and Dolores Wallaces, who provided substantive technical comments on the draft manuscript as part of the rigorous NIST process for reviewing technical and scientific manuscripts prior to submitting them for consideration by conferences and journals. We also appreciate the encouragement provided by David Tennenhouse, who reminded us of our responsibility to share the results of our work. Finally, if readers find the manuscript to be clear and interesting, then much of the credit belongs to the anonymous reviewers whose careful reading and significant suggestions showed us how to improve the paper. Of course, we take responsibility for any errors or flaws remaining in the paper.

REFERENCES

BLACKWELL, A. F. 1996. Meta-cognitive theories of visual programming: What do we think we are doing? In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, 240–246.

BOLOIX, G., SORENSON, P. G., AND TREMBLAY, J. P. 1992. Transformations using a meta-system approach to software development. *Software Engineering Journal* (Nov.), 425–437.

BOOCH, G. 1986. Object-oriented development. *IEEE Transactions on Software Engineering* (Feb.), 211–221.

BOOCH, G. 1991. *Object Oriented Design With Applications*. Benjamin/Cummings, Redwood City, California.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 1999. *The Unified Modeling Language User Guide*. Addison Wesley, Reading, Mass.

BRACHMAN, R. J. AND SCHMOLZE, J. G. 1989. An overview of the KL-ONE knowledge representation system. In *Readings in Artificial Intelligence and Databases*, J. Mylopoulos and M. L. Brodie, Eds. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 207–229.

COAD, P. AND YOURDON, E. 1991. *Object-Oriented Analysis*. Yourdon Press, Englewood Cliffs, New Jersey.

DEMARCO, T. 1978. *Structured Analysis and System Specification*. Prentice-Hall, Englewood Cliffs, New Jersey.

FIKES, R. AND KEHLER, T. 1985. The role of frame-based representation in reasoning. *Communications of the ACM 28*, 9, 904–920.

FOWLER, M. 1997. *UML Distilled*, (with Kendall Scott), Addison-Wesley.

GOMAA, H. 1993. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley Publishing Company, Reading Massachusetts.

GOMAA, H. 2000. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley Publishing Company, Reading, Massachusetts.

HAARSLEV, V. AND WESSEL, M. 1996. GenEd—An editor with generic semantics for formal reasoning about visual notations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, 204–212.

HAREL, D. 1988. On visual formalisms. *Communications of the ACM 31*, 5, 514–530.

HAREL, D. AND GARY, E. 1996. Executable object modeling with statecharts. In *Proceedings of the ACM/IEEE 18th International Conference on Software Engineering*.

HATLEY, D. AND PIRBHAI, I. 1988. *Strategies for Real-Time System Specification*. Dorset House, New York.

HAYES-ROTH, F. 1985. Rule-based systems. *Communications of the ACM 28*, 9, 921–932.

JACOBSON, I., BOOCH, G., AND RUMBAUGH, J. 1999. *The Unified Software Development Process*. Addison Wesley, Reading, Mass.

JACKSON, M. 1983. *System Development*. Prentice-Hall, Englewood Cliffs, NJ.

KARIMI, J. AND KONSYNSKI, B. R. 1988. An automated software design assistant. *IEEE Transactions on Software Engineering* (Feb.), 194–210.

LIM, E. AND CHERKASSKY, V. 1992. Semantic networks and associative databases. *IEEE Expert* (Aug.), 31–40.

LOR, K. E. AND BERRY, D. M. 1991. Automatic synthesis of SARA design models from systems requirements. *IEEE Transactions on Software Engineering* (Dec.), 1229–1240.

MICHALSKI, R. S. 1980. Pattern recognition as rule-guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence 2*, 4.

MELLOR, S. J. AND WARD, P. T. 1986. *Structured Development for Real-Time Systems*, Vol. 3, *Implementation Modeling Techniques*. Yourdon Press, Englewood Cliffs, NJ.

MILLS, K. 1996. Automated generation of concurrent designs for real-time software. Ph.D. dissertation, George Mason University.

MILLS, K. AND GOMAA, H. 1996. A knowledge-based approach for automating a design method for concurrent and real-time systems. In *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering*.

NASA. 1993. CLIPS reference manual. Three Volumes, CLIPS Version 6.0, NASA, Software Technology Branch.

NIELSEN, K. AND SHUMATE, K. 1987. Designing large real-time systems with Ada. *Communications of the ACM 30*, 8, 695–715.

NIELSEN, K. AND SHUMATE, K. 1988. *Designing Large Real-Time Systems with Ada*. McGraw-Hill, New York.

PUIGSEGUR, J., AGUSTI, J., AND ROBERTSON, D. 1996. A visual logic programming language. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. 1991. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey.

RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. 1999. *The Unified Modeling Language Reference Manual*. Addison Wesley, Reading, Mass.

SANTUCCI, G. 1996. On graph-based interaction for semantic query languages. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, 76–83.

SHLAER, S. AND MELLOR, S. J. 1992. *Object Lifecycles—Modeling the World in States*. Yourdon Press, Englewood Cliffs, New Jersey.

TSAI, J. P. AND RIDGE, J. C. 1988. Intelligent support for specifications transformation. *IEEE Software* (Nov.), 28–35.

WARD, P. AND MELLOR, S. 1985. *Structured Development for Real-Time Systems*. Four Volumes, Prentice-Hall, Englewood Cliffs, New Jersey.

YOURDON, E. AND CONSTANTINE, L. L. 1979. *Structured Design*. Prentice-Hall, Englewood Cliffs, New Jersey.

Received March 1998; revised October 1998 and March 2000; accepted April 2000